In this assignment, you will implement the game of Boggle^{TM}, which will require you to (1) design and implement a number of recursive algorithms and (2) think about various implementation details for the Boggle dictionary.

1   The Game of Boggle

Boggle is a word game played with sixteen cubes, where each side of each cube has one letter of the alphabet. The cubes are randomly arranged on a 4×4 grid, with one legal configuration shown below:

E	E	C	A
A	L	E	P
H	N	B	O
Q	T	T	Y

The object of the game is to identify—in this grid of letters—words that satisfy the following conditions:

- The word must be at least four letters long.

- The path formed by the sequence of letters in the word must be connected horizontally, vertically, or diagonally.

- For a given word, each cube may only be used once.

For example, the above board contains the the word PEACE, which is legally connected as shown below.

E	E←C←A		
A	L	E←P	
H	N	B	O
Q	T	T	Y

The board does not contain the word PLACE, because the L and the P are not connected, and the board does not contain the word POPE, because the same P cannot be used more than once for a given word.

Points are scored based on the length of the word. Four-letter words are worth 1 point, five-letter words are worth 2 points, etc.

2   Your Boggle Game

You will create a Boggle game which randomly sets up a board. Your game will then allow a human to identify a list of words in the board, which your program will verify against some dictionary and for which your program will compute a score. When the human can think of no more words, your program will create a list of legal words that the human did not identify.

For example, if the human identified the following words for the above board:

lean   pace   bent   peel   pent   clan   clean   lent

Your program would verify that each of these words was legal and would assign a score of 9 for the seven four-letter words and one five-letter word. Your program would then use a dictionary, whose words we will provide, to identify the following words (yeah, we didn't know that hant and blent were words, either; neither does the Unix spellchecker):

elan	celeb	cape	capelan	capo	cent	cento	alee
alec	anele	leant	lane	leap	lento	peace	pele
penal	hale	hant	neap	blae	blah	blent	becap
benthal	bott	open	thae	than	thane	toecap	tope
topee	toby						

For this wondrous list of words, your program would obtain a score of 53, thoroughly embarrassing the feeble human player.

2.1 Initializing the Game

Our implementation of Boggle will support any board size, represented as an input variable n. The game uses n^2 cubes, each with a particular set of letters on it. These letters have been chosen so that common letters are more likely to appear, and so that there is a good mix of consonants and vowels. You should initialize your cubes from the file, `cubes.txt`, which contains the following data:

```
LRYTTE
VTHRWE
EGHWNE
SEOTIS
ANAEEG
IDSYTT
OATTOW
MTOICU
AFPKFS
XLDERI
HCPOAS
ENSIEU
YLDEVR
ZNRNHL
NMIQHU
OBBAOJ
```

Each line represents the six characters that will appear on the faces of a single cube. To initialize your game, you should read this file and store the data in a data structure that represents the n^2 cubes. You can always assume there will be $\geq n^2$ non-empty lines in this cubefile, in which you should use the first n^2.

For each game, your program should randomly shuffle each cube and randomly distribute the cubes amongst the $n \times n$ grid. There are many ways to do this. You could lay down the cubes and start swapping them around, or you could pick cubes randomly and lay them down one at a time. We provide code that rolls and distributes the dice.

Your program will need to implement a simple dictionary that can read a large number of words (hundreds of thousands to millions of words) and store it in a Trie. The dictionary file is called `words.txt`, and it contains one word per line, with the words in ascending lexicographic order.

A Trie is a 26-ary tree—that is, each node has 26 children, one per letter that could follow the current letter. The root node is the empty string, and by traversing down the Trie, you iterate over a list of valid words. Because a word can be a prefix of a word and a word itself (for instance, "pot" is a word and a prefix of "potato"), each node needs to remember if it is the end of a word.

Once your initialization is complete, you're ready to implement two types of recursive search, one for the human player and one for the computer. Each search uses a distinct type of recursion. For the human, you search for a specific word and stop as soon as it's found in the board, while for the computer, you are searching for all possible words. You might be tempted to integrate the two types of recursion into a single routine, but this will be unnecessarily complex, so we advise you to resist this temptation. However, the two routines may well share helper methods.

2.2 User Interface (UI)

When the game starts, the UI creates and displays a new scrambled Boggle board. You may also specify an explicit board using command line arguments. It then accepts a word from the human player as input, checks the word for validity, and then computes a score for the word. If the word is valid, the board prints again with your word capitalized.

If the word is too short, not on the board, not a legal word, or is already used by the player, an error message is printed. The player does not get credit for such words.

After the human indicates that they are done, the computer player gets to select all valid words that were not identified by the human. The program then displays the respective scores. After every game, the UI prompts the user for another game and acts accordingly.

The game is implemented in `boggle.py` in the top-level directory. It should run with `python3 boggle.py`. Use `python3 boggle.py -h` for a full list of options.

2.3   The BoggleGame Interface

Everything that manages the mechanics of the game should be contained in a class that implements the BoggleGame interface. This interface provides all the necessary functions for implementing a basic generalized game of Boggle. The use of this interface completely separates the code that manages the game from the code that implements the UI.

```
class BoggleGame(abc.ABC):
    class SearchTactic(enum.Enum):
        SEARCH_BOARD = enum.auto()
        SEARCH_DICT = enum.auto()

    def new_game(
        self, size: int, cubefile: str, dict: BoggleDictionary
    ) -> None: ...
    def get_board(self) -> List[List[str]]: ...
    def add_word(self, word: str) -> int: ...
    def get_last_added_word(self) -> Optional[List[Tuple[int, int]]]: ...
    def set_game(self, board: List[List[str]]) -> None: ...
    def get_all_words(self) -> Collection[str]: ...
    def set_search_tactic(self, tactic: SearchTactic) -> None: ...
    def get_score(self) -> int: ...
```

Implement your game engine as `class MyGameManager` **in** `my_game_manager.py`.
For additional details, see `boggle_game.py`.

2.4   Dictionary Interface

To check for legal words, your program will need to search the dictionary, which you will implement. The methods in the `BoggleDictionary` interface, shown below, allow you to create a new dictionary and to insert words into it as an entire collection stored in a single file. The interface also specifies methods to determine whether a string is found in the dictionary or whether it is a prefix of a word in the dictionary. These methods must be case-insensitive. Finally, the dictionary interface is an `Iterable` type, so you will need to implement the `__iter__(self)` method in your dictionary class. You will need to return an `Iterator` object from the dictionary class's `__iter__(self)` method. For us to grade your assignment, you should ensure that your iterator terminates properly. For more information about the Iterable and Iterator types, see the following:
- The official documentation:
    - `https://docs.python.org/3/glossary.html#term-iterable`
    - `https://docs.python.org/3/library/stdtypes.html#typeiter`
- Additional resources:
    - `https://wiki.python.org/moin/Iterator`
    - `https://www.programiz.com/python-programming/iterator`

Although reading the official documentation is not always the easiest way to learn how to accomplish a particular task, it contains a great deal of technical information that is necessary to understand how the language works.

```
class BoggleDictionary(abc.ABC, Iterable):
    def load_dictionary(self, filename: str) -> None: ...
    def is_prefix(self, prefix: str) -> bool: ...
    def contains(self, word: str) -> bool: ...
    def __iter__(self) -> Iterator[str]: ...
```

There are interesting design issues associated with the dictionary. A Trie is a good fit because lookup operations do not necessarily take longer on a longer dictionary. **Your dictionary should be implemented as** `class`

`TrieDictionary` **in** `trie_dictionary.py`.
For additional details, see `boggle_dictionary.py`.

2.5 Searching for Words

For the computer's turn, your job is to find all words that the human player missed. In this phase, the same conditions apply as for the human's turn, plus the additional restriction that the computer cannot include any words that were already found by the human.

To do this, you will need to search the Boggle board for all words present. The `board_driven_search()` and `dictionary_driven_search()` methods in the game interface should implement one of the two following search strategies.

2.5.1 Board-Driven Search

The first strategy is the obvious one: Recursively search the board for words beginning at each square on the board. As with any exponential search, you should prune the search as much as possible to speed things up. One important strategy is to recognize when you're going down a dead end. For example, if you are searching for words that begin with the letters "ZX", you can use the dictionary's `is_prefix()` method (which you will implement), to determine that there are no English words that begin with this prefix and to recognize that your program can abandon this search path.

2.5.2 Dictionary-Driven Search

You should also implement a second strategy that iterates over all words in the Dictionary and checks whether these words can be found on the given board. There are various tricks you can play to improve the efficiency of this approach. We leave it to you to find these tricks.

2.6 Correctness

The BoggleGame interface allows the game to be developed separately from user interface considerations. We should be able to use your game in our UI or test program without modification. The interface also provides the `set_game()` method, which is *extremely* useful for debugging and testing your search functions.

Your user interface is allowed to assume traditional Boggle rules, although you might wish to make it somewhat more general. Your game and dictionary should not assume these things.

You may want to test Boggle at different sizes and with different data to make sure you aren't making hidden assumptions.

We provide most of the BoggleGame implementation in `my_game_manager.py`. You do not need to change any of the implemented functions, but you are free to do so.

3 What to Turn In

Use the **Gradescope** page of the **Canvas** website to submit your repository via a linked Github account. Please refer to the "Submission Instructions" page on edx for Gradescope submission instructions. Canvas is available at `http://canvas.utexas.edu/`.

Make sure all classes are in the correct files. Also make sure that your submission is using the correct directory structure. (The same directory structure provided by the starter code.)

Important Details.
- This assignment is due at **11:59pm** on the due date. Late assignments will be penalized 20% per day (unless your Slip days are assigned to it at the end of the semester).
- You may work in a group of up to 3 students, but make 1 submission per person.

4 Karma

If you have time on your hands, you might consider the question, "What board configuration produces the highest score?" and you might consider some related questions: How would you identify such a board configuration? Empirically? Theoretically? How many possible configurations are there?